The DeCCo project papers IV

Z Specification of Linker and Hexer

Susan Stepney

University of York Technical Report YCS-2002-361

June 2003

Contents

Preface

Historical background of the DeCCo project

In 1990 Logica's Formal Methods Team performed a study for RSRE (now QinetiQ) into how to develop a compiler for high integrity applications that is itself of high integrity. In that study, the source language was Spark, a subset of Ada designed for safety critical applications, and the target was Viper, a high integrity processor. Logica' Formal Methods Team developed a mathematical technique for specifying a compiler and proving it correct, and developed a small proof of concept prototype. The study is described in [Stepney *et al.* 1991], and the small case study is worked up in full, including all the proofs, in [Stepney 1993]. Experience of using the PVS tool to prove the small case study is reported in [Stringer-Calvert *et al.* 1997]. Futher developments to the method to allow separate compilation are described in [Stepney 1998].

Engineers at AWE read about the study and realised the technique could be used to implement a compiler for their own high integrity processor, called the ASP (Arming System Processor). They contacted Logica, and between 1992 and 2001 Logica used these techniques to deliver a high integrity compiler, integrated in a development and test environment, for progressively larger subsets of Pascal.

The full specifications of the final version of the DeCCo compiler are reproduced in these technical reports. These are written in the Z specification language. The variant of Z used is that supported by the *Z Specific Formaliser* tool [Formaliser], which was used to prepare and type-check all the DeCCo specifications. This variant is essentially the Z described in the *Z Reference Manual* [Spivey 1992] augmented with a few new constructs from *ISO Standard Z* [ISO-Z]. Additions to ZRM are noted as they occur in the text.

The DeCCo Reports

The DeCCo Project case study is detailed in the following technical reports (this preface is common to all the reports):

I. **Z Specification of Pasp**
The denotational semantics of the high level source language, Pasp. The definition is split into several static semantics (such as type checking) and a dynamic semantics (the meaningof executing a program). Later smeantics are not defined for those programs where the result of earlier semantics is error.

II. **Z Specification of Asp, AspAL and XAspAL**
The denotational semantics of the low level target assembly languages. XAspAL is the target of compilation of an individual Pasp module; it is AspAL extended with some cross-module instructions that are resolved at link time. The meaning of these extra instructions is given implicitly by the specification of the linker and hexer. AspAL is the target of linking a set of XAspAL modules, and also the target of compilation of a complete Pasp program. Asp is the non-relocatable assembly language of the chip, with AspAL's labels replaced by absolute program addresses. The semantics of programs with errors is not defined, because these defintions will only ever be used to define the meaning of correct, compiled programs.

III. **Z Specification of Compiler Templates**
The operational semantics of the Pasp source language, in the form of a set of XAspAL target language templates.

IV. **Z Specification of Linker and Hexer**
The linker combines compiled XAspAL modules into a single compiled AspAL program. The hexer converts a relocatable AspAL program into an Asp program located at a fixed place in memory.

V. **Compiler Correctness Proofs**
The compiler's operational semantics are demonstrated to be equivalent to the source language's denotational semantics, by calculating the meaning of each Pasp construct, and the corresponding meaning

of the AspAL template, and showing them to be equivalent. Thus the compiler transformation is *meaning preserving*, and hence the compiler is correct.

VI. **Z to Prolog DCTG translation guidelines**
The Z specifications of the Pasp semantics and compiler templates are translated into an executable Prolog DCTG implementation of a Pasp interpreter and Pasp-to-Asp compiler. The translation is done manually, following the stated guidelines.

Acknowledgements

We would like to thank the client team at AWE – Dave Thomas, Wilson Ifill, Alun Lewis, Tracy Bourne – for providing such an interesting development project to work on.

We would like to thank the rest of the development team at Logica: Tim Wentford, John Taylor, Roger Eatwell, Kwasi Ametewee.

1 Introduction

This document provides the specification for the linker and hexer.

The linker transforms a sequence of XAspAL modules into an AspAL program. The linker specification includes the abstract syntax and semantics, and links together a number of compiled AspAL modules, *CompModule*. The linker output program can be demonstrated to have the same meaning as an equivalent Pasp program.

The hexer transforms an AspAL program into an Asp progrm. This transformation is straightforward for the majority of the instructions, where no change is required. AspAL label instructions are transformed to Asp absolute program address instructions. This can result in a single AspAL instruction being transformed in to a sequence of Asp instructions.

2 Informal overview of linker

This section gives an overview of the definition of the linker. The purpose of the linker is to link compiled module segments into one compiled program.

2.1 Design Decisions

There are a number of design decisions that have been taken during the specification of the linker, and modules. These are as follows

- All module names have to be unique.
- All export identifiers have to be unique – that is, procedures and functions with the same name cannot be exported from two different modules.
- All standard modules have to have an export declaration, even if it is empty.
- All names of functions and procedures imported into a module must be exported in a previous module in the linking list.
- No import can also be exported from the same module.

2.2 Concepts

The modules are written in Pasp. Pasp modules are compiled into XAspAL code segments. The XAspAL instruction set is the labelled AspAL instruction set extended with extra X instructions required by the linker.

The set of instructions constituting AspAL is the XAspAL set with the cross-module instructions removed.

$$AspAL == XAspAL \setminus \{xProc, xCall, xBvar\}$$

The Asp instruction set has jump and goto instructions to specific data address locations. It is AspAL with all the remaining X-instructions removed, and the label values replaced by the specific locations and offsets.

$$Asp == AspAL \setminus \{xData, xIndirect, xLabel\}$$

So, for a Pasp module, the compiler performs the following action

$$compile : Module \pfun CompModule$$

which is

$$compile : Module \pfun \seq XAspAl \times \ldots$$

One the Pasp modules havve been compiled individually, they can be linked (see later).

2.3 Compiler output

The output from the compilation stage is a collection of items. These are required by the linker, and comprise the module's name, the compiled XAspAl instructions, the amount of memory required for stack and heap, and the addresses of the fixed variables, the symbol declaration environment, the type checking environment, and the compiler translation environment. These are combined into a single syntactic structure *CompModule*, which has been declared earlier but is displayed again here for convenience, along with definitions of the module compiler templates.

$$
\begin{array}{l}
CompModule == \\
\quad \begin{array}{ll}
ID & \text{module name} \\
\times \seq X\_INSTR & \text{compiled code, XAspAL} \\
\times \nat & \text{size, heap + stack} \\
\times (\power ATTR \pfun \power DATA\_ADDRESS) & \text{fixed AT vars} \\
\times EnvD & \text{declaration environment} \\
\times EnvT & \text{type environment} \\
\times EnvOTrace & \text{translation environment} \\
\times MO & \text{memory allocation map}
\end{array}
\end{array}
$$

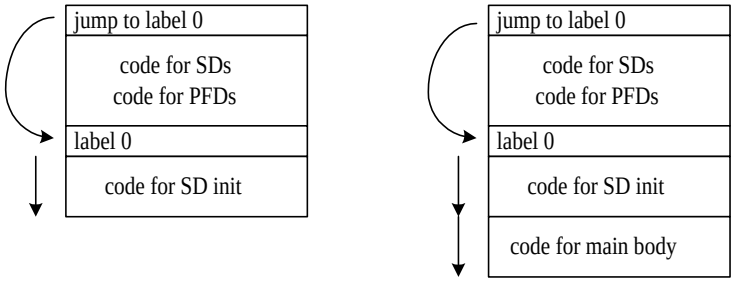

Figure 1: Compiled code structure of a module, and a main module

$$\mathcal{O}_M : Module \pfun CompModule$$
$$\mathcal{O}_{M^*} : \mathrm{seq}\, Module \pfun \mathrm{seq}\, CompModule$$
$$\mathcal{O}_{MA} : MainModule \pfun CompModule$$

The linker then takes these segments to AspAL (which is a subset of X_INSTR as defined previously). The standard module and main module are treated differently, and so are defined separately.

$$link : \mathrm{seq}\, CompModule \times CompModule \pfun \mathrm{seq}\, X\_INSTR$$

Then the hexer takes the AspAL, and generates Asp. It does this by removing the abstract jumps to labels, replacing them with jumps to specific addresses, thereby removing the relocatable nature of the AspAL.

$$hex : \mathrm{seq}\, AspAL \pfun \mathrm{seq}\, Asp$$

2.4 Code structure

Figure 1 shows the structure of a compiled module, and a compiled main module.

Figure 2 shows the structure of a linked program of modules. The linker fixes up the labels so that they do not clash, and puts the modules in contiguous memory space, so that at the end of initialisation code in one module is the jump at the beginning of the next module.

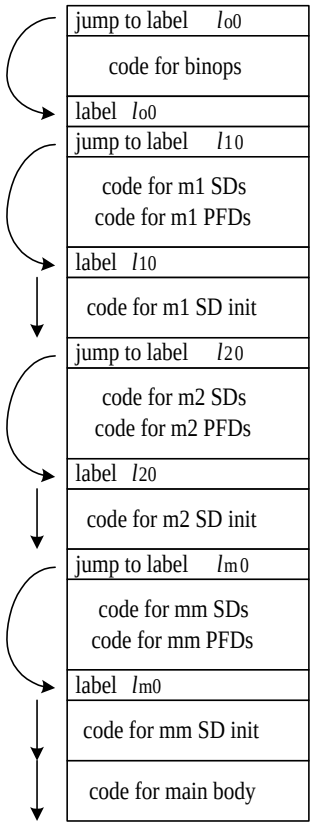

Figure 2: Linked code structure of a sequence of modules

3 Linker input — assigning memory locations

Parent Section: Compiler Specification

The input to the linker, is a sequence of *CMODULE*, along with some other required information. The *minaddr*, and *maxaddr* data addresses specify the memory range where the linked program's data should reside. All the fixed variables should be within this range, as well as the modules' heaps and stacks and the operator stack. In addition, the value of *sizetopop* is required, which is currently fixed at 16.

The assignment of memory addresses for each module, fixed variable, and operator stack has to be performed in a definite order, and at each stage checks are required to ensure that the space has been correctly assigned.

3.1 Memory Assignment

The free (unallocated) memory is modelled as a sequence of free memory blocks (pairs of addresses). The pairs of addresses define a range of memory locations, inclusive at both extremities.

$$\begin{aligned} Free == \{\ & F : \mathrm{seq}(ADDR \times ADDR) \mid \\ & (\ \forall n : \mathrm{dom}\, F \bullet (F\ n).1\ \leq (F\ n).2\ ) \\ & \land\ (\ \forall n : \mathrm{dom}(front\ F) \bullet (F\ n).2 < (F\ (n+1)).1\ )\ \} \end{aligned}$$

The constraints ensure that no blocks are empty, that blocks do not overlap, and that later blocks are at higher addresses.

When a new address is required a value is removed from those available. Initially *Free* is given by the hardware memory map.

Address assignment proceeds as follows:

1. The given location 'AT' variables are assigned.
2. The operator stack is assigned at the highest value possible, and the value of *topOp* is set

3. Each module's space is assigned in order, using the lowest addressed large enough block.

3.2 Free block manipulations

freesize takes a sequence of free blocks, and returns the corresponding sequence of their sizes.

$$freeSize : Free \rightarrow \operatorname{seq} \mathbb{N}$$

$$\forall F : Free \bullet freeSize\ F = \{\ n : \operatorname{dom} F \bullet n \mapsto (1 + (F\ n).2 - (F\ n).1)\ \}$$

blockify takes a set of addresses, and returns the free block structure that contains precisely those addresses.

$$blockify : \mathbb{P}\, ADDR \rightarrow Free$$

$$\forall \Delta : \mathbb{P}\, ADDR \bullet \exists F : Free \mid \Delta = \bigcup \{\ n : \operatorname{dom} F \bullet (F\ n).1 \mathinner{..} (F\ n).2\ \} \bullet blockify\ \Delta = F$$

3.3 Extractor functions

Two functions are required to extract required information from the *CMODULE*s. These are for the variables at fixed addresses, and the sizes of the modules.

The *getAts* function extracts the fixed 'AT' addresses from a sequence of modules.

$$getAts : \operatorname{seq} CompModule \nrightarrow \operatorname{seq}(\mathbb{P}\, ATTR \nrightarrow \mathbb{P}\, ADDR)$$

$$\forall M : \operatorname{seq} CompModule \bullet getAts\ M = \{\ n : \operatorname{dom} M \bullet n \mapsto (M\ n).\alpha\ \}$$

The *getModSize* function extracts the module sizes from a sequence of modules.

$$
\begin{array}{|l}
getModSize : \mathrm{seq}\, CompModule \pfun \mathrm{seq}\, \mathbb{N} \\
\hline
\forall\, M : \mathrm{seq}\, CompModule \bullet \\
\quad getModSize\ M = \{\ n : \mathrm{dom}\, M \bullet n \mapsto (M\ n).size\ \}
\end{array}
$$

3.4 Allocation of operator stack

The operator stack memory is allocated at the top of the highest possible location in the given free memory (done after assignment of the AT variables).

$$
\begin{array}{|l}
assignTopOp : Free \times \mathbb{N} \pfun Free \times ADDR \\
\hline
\forall\, F : Free;\ size : \mathbb{N} \bullet \\
\quad \exists\, s : \mathrm{seq}\, \mathbb{N};\ maxBlock : \mathbb{N} \mid \\
\qquad s = freeSize\ F \land maxBlock = max(\mathrm{ran}\, s) \bullet \\
\qquad (maxBlock < size \Rightarrow assignTopOp(F, size) = (F, 0)) \\
\qquad \land\ (size \leq maxBlock \Rightarrow \\
\qquad\quad (\ \exists\, n : \mathrm{dom}\, F;\ F' : Free;\ \delta : ADDR \mid \\
\qquad\qquad n = max(\mathrm{dom}(s \rhd (0\,..\,(size - 1)))) \\
\qquad\qquad \land\ (F', \delta) = \\
\qquad\qquad\quad \textbf{if}\ s\ n = size \\
\qquad\qquad\quad \textbf{then}\ (squash(\{n\} \lhd F), (F\ n).1) \\
\qquad\qquad\quad \textbf{else}\ (F \oplus \{n \mapsto ((F\ n).1, (F\ n).2 - size)\}, \\
\qquad\qquad\qquad\quad (F\ n).2 - size + 1) \bullet \\
\qquad\qquad assignTopOp(F, size) = (F', \delta))\ )
\end{array}
$$

- $maxblock < size$: there is no free block large enough for the operator stack.
- n : the index of the highest block in memory large enough to hold the operator stack
- F' : the free memory after assigning the operator stack: if the selected block is the same size as the stack, it is removed from the free sequence; if it is larger than the operator stack, it remains in the free sequence, but with reduced size.

- δ : the address allocated to of the operator stack (the smallest data address of the assigned block).

3.5 Assignment of module memory space

Each module is allocated its required space (the total of its stack and heap). The stack starts at the top of the allocated block of memory, and works down. The heap upwards from the bottom.

The *assignVal* function is very similar to *assignTopOp*, except that it starts at the lowest free memory address trying to find space to allocate the memory for the module. If no block of memory large enough can be found then the module has the address start zero assigned to it.

$$
\begin{array}{|l}
assignVal : Free \times \mathbb{N} \pfun Free \times ADDR \\
\hline
\forall\, F : Free;\ size : \mathbb{N} \bullet \\
\quad \exists\, s : \operatorname{seq} \mathbb{N};\ maxBlock : \mathbb{N} \mid \\
\qquad s = freeSize\ F \land maxBlock = max\ (\operatorname{ran} s) \bullet \\
\qquad (maxBlock < size \Rightarrow assignVal(F, size) = (F, 0)) \\
\qquad \land\ (size \leq maxBlock \Rightarrow \\
\qquad\quad (\ \exists\, n : \operatorname{dom} F;\ F' : Free;\ \delta : ADDR \mid \\
\qquad\qquad n = min(\operatorname{dom}(s \rhd (0 \mathinner{..} (size - 1)))) \\
\qquad\qquad \land\ (F', \delta) = \\
\qquad\qquad\quad \textbf{if}\ s\ n = size \\
\qquad\qquad\quad \textbf{then}\ (squash(\{n\} \ndres F), (F\ n).1) \\
\qquad\qquad\quad \textbf{else}\ (F \oplus \{n \mapsto ((F\ n).1 + size, (F\ n).2)\}, \\
\qquad\qquad\qquad\quad (F\ n).1) \bullet \\
\qquad\qquad assignVal(F, size) = (F', \delta))\ )
\end{array}
$$

The *assignMod* function takes a sequence of free blocks, and a sequence of module sizes, and assigns each module the required space, if possible. The output is the updated sequence of free blocks and a sequence of start addresses for each module. Any modules that cannot be allocated space are assigned the address 0.

$$
\begin{array}{|l}
assignMod : Free \times \mathrm{seq}_1 \mathbb{N} \pfun Free \times \mathrm{seq}_1 ADDR \\
\hline
\forall F : Free;\ size : \mathbb{N} \bullet \\
\quad \exists F' : Free;\ \delta : ADDR \mid \\
\qquad (F', \delta) = assignVal(F, size) \bullet \\
\qquad assignMod(F, \langle size \rangle) = (F', \langle \delta \rangle) \\
\forall F : Free;\ size : \mathbb{N};\ s : \mathrm{seq}_1 \mathbb{N} \bullet \\
\quad \exists F', F'' : Free;\ \delta : ADDR;\ \Delta : \mathrm{seq}_1 ADDR \mid \\
\qquad (F', \delta) = assignVal(F, size) \\
\qquad \land\ (F'', \Delta) = assignMod(F', s) \bullet \\
\qquad assignMod(F, \langle size \rangle \frown s) = (F'', \langle \delta \rangle \frown \Delta)
\end{array}
$$

3.6 Additional global functions required

Additional global functions and declarations are required to resolve actual parameter addressing across modules.

A sequence of the addresses of the *at* variables is required. This is made global so that it can be accessed from within the type checking and the linker; its value is defined in $\mathcal{T}_L$ later.

$$
\begin{array}{|l}
atsMap : \mathrm{seq}(\mathbb{P}\, ATTR \pfun \mathbb{P}\, ADDR)
\end{array}
$$

Functions that map each module to its start address, to its operational trace environment, and to its memory allocation map, are required. A value for the address of *topop* is required. These are made global so that they can be accessed from within the linker; their values are defined in *link* later.

$$
\begin{array}{|l}
modStart : ID \pfun ADDR \\
compEnv : ID \pfun EnvOTrace \\
compMap : ID \pfun MO \\
topop : ADDR
\end{array}
$$

4 Linker symbol declaration semantics

The symbol declaration semantics for the linker are analogous to those defined for the modularised program, $\mathcal{D}_{MP}$.

All the symbols within each module have been checked on compilation. Each module's symbol declaration semantics has been restricted to those *ID*s visible outside the module, that is, exports and module names. We therefore want to ensure that module names are not multiply declared. We also want to check that imports to a module have been declared as an export in a previous module. Therefore, each of the symbol declaration semantics also includes all the imported *ID*, so that these can be checked for previous export declarations.

The symbol environment defined for the modularised program semantics, *EnvD*, is used for the linker also. The function *checkLink* is re-used to perform the required linker checks.

The *CompModule*'s declaration semantics update the linker environment *EnvD* with the appropriate information from the module's declaration semantics. The *EnvD* environment is updated with *checkLink* applied to the module's environment.

$$\mathcal{D}_{LM} : CompModule \nrightarrow EnvDTrace \nrightarrow EnvDTrace$$

$$\forall\, CompModule \bullet \mathcal{D}_{LM}\ \theta CompModule = checkLink(\xi, \rho\delta)$$

The usual extension to a sequence of modules allows a definition of $\mathcal{D}_{LM^*}$.

$$\mathcal{D}_{LM^*} : \mathrm{seq}\, CompModule \nrightarrow EnvDTrace \nrightarrow EnvDTrace$$

$$\mathcal{D}_{LM^*}\langle\,\rangle = \mathrm{id}\, EnvDTrace$$

$$\forall\, cm : CompModule;\ M : \mathrm{seq}\, CompModule;\ \rho\delta t : EnvDTrace \bullet$$
$$\exists\, \rho\delta t', \rho\delta t'' : EnvDTrace \mid$$
$$\rho\delta t' = \mathcal{D}_{LM}\ cm\ \rho\delta t \land \rho\delta t'' = \mathcal{D}_{LM^*}\ M\ \rho\delta t' \bullet$$
$$\mathcal{D}_{LM^*}(\langle cm\rangle\ ^\frown M)\rho\delta t = \rho\delta t''$$

The $\mathcal{D}_L$ semantics checks naming for a complete program. This ensures that there are no name clashes between any exports, or module names, as well

as checking that for each import there is a preceding export from another module.

$$
\begin{array}{|l}
\mathcal{D}_L : \operatorname{seq} \mathit{CompModule} \times \mathit{CompModule} \pfun \mathit{EnvDTrace} \\
\hline
\forall\, M : \operatorname{seq} \mathit{CompModule};\ cm : \mathit{CompModule} \bullet \\
\quad \exists\, \rho\delta t, \rho\delta t' : \mathit{EnvDTrace} \mid \\
\qquad \rho\delta t = \mathcal{D}_{LM^*}\ M\ \varnothing \land \rho\delta t' = \mathcal{D}_{LM}\ cm\ \rho\delta t \bullet \\
\qquad \mathcal{D}_L(M, cm) = \rho\delta t' \\
\end{array}
$$

The linker's declaration semantics are correct if the following condition is satisfied.

LinkDeclOkay

$$
\begin{array}{|l}
M : \operatorname{seq} \mathit{CompModule} \\
cm : \mathit{CompModule} \\
\hline
\rho\delta t0 = \mathcal{D}_L(M, cm) \\
(\operatorname{ran} \circ \bigcup \circ \operatorname{ran})\rho\delta t0 \subseteq \{\mathit{checkOK}, \mathit{ImportOK}\} \\
\end{array}
$$

This check encapsulates all the modules' and program's declaration checking. If this fails then the particular module, and identifier can be found which are incorrectly declared.

5 Linker type checking semantics

The type checking semantics for the linker are analogous to those defined for the modularised program, $\mathcal{T}_{MP}$.

We know, from the symbol declaration semantics for the linker, that any import declaration is previously exported. The main type checking for the linker is to ensure that the types of imported declarations match those of the actual declaration in the exporting module. A type check environment is produced that defines the check state for each import to modules within the program.

The *CompModule*'s type semantics update the linker's type check environment with a check state for each imported declaration to the module. It is already known from checks at the compilation stage that the type check environment for the module checks correctly.

$$\mathcal{T}_{LM} : CompModule \pfun EnvTTrace \times EnvDTrace \pfun EnvTTrace \times EnvDTrace$$

$$\forall\, CompModule \bullet \mathcal{T}_{LM}\ \theta CompModule = checkLinkType(\xi, \rho\tau)$$

The usual extension to a sequence of modules allows a definition of $\mathcal{T}_{LM^*}$.

$$\mathcal{T}_{LM^*} : \mathrm{seq}\, CompModule \pfun EnvTTrace \times EnvDTrace \pfun EnvTTrace \times EnvDTrace$$

$$\mathcal{T}_{LM^*}\langle\,\rangle = \mathrm{id}(EnvTTrace \times EnvDTrace)$$

$$\forall\, cm : CompModule;\ M : \mathrm{seq}\, CompModule;$$
$$\rho\tau\delta t : EnvTTrace \times EnvDTrace \bullet$$
$$\exists\, \rho\tau\delta t', \rho\tau\delta t'' : EnvTTrace \times EnvDTrace \mid$$
$$\rho\tau\delta t' = \mathcal{T}_{LM}\ cm\ \rho\tau\delta t \land \rho\tau\delta t'' = \mathcal{T}_{LM^*}\ M\ \rho\tau\delta t' \bullet$$
$$\mathcal{T}_{LM^*}(\langle cm\rangle \frown M)\rho\tau\delta t = \rho\tau\delta t''$$

The $\mathcal{T}_L$ semantics type checks the linking of a complete program. The module's type checking is performed for the compilation, and is known to be correct. The linker semantics ensure that the types of declarations that are exported from a module, and any import using that declaration, are the same. This also sets the global *atsMap* sequence.

$$
\begin{array}{|l}
\mathcal{T}_L : \operatorname{seq} CompModule \times CompModule \pfun EnvTTrace \times EnvDTrace \\
\hline
\forall\, M : \operatorname{seq} CompModule;\ cm : CompModule \mid LinkDeclOkay \bullet \\
\quad \exists\, \rho\tau\delta t, \rho\tau\delta t' : EnvTTrace \times EnvDTrace \mid \\
\qquad atsMap = getAts(M \frown \langle cm \rangle) \\
\qquad \land\ \rho\tau\delta t = \mathcal{T}_{LM^*}\ M(\varnothing, \varnothing) \\
\qquad \land\ \rho\tau\delta t' = \mathcal{T}_{LM}\ cm\ \rho\tau\delta t \bullet \\
\quad \mathcal{T}_L(M, cm) = \rho\tau\delta t'
\end{array}
$$

The type checking semantics also ensures that attributed variables are assigned to sensible memory locations. These locations depend on the supplied hardware memory map. *minAddr* and *maxAddr* are the lowest and highest addressable memory locations. *voidMap* is the set of non-addressable locations between these extremes; *nvramMap* is the set of *nvram* locations; *readOnlyMap* is the set of read only locations; *writeOnlyMap* is the set of write only locations. *ram* is all the remaining locations, derived from the above values.

$$
\begin{array}{|l}
\hline
HwMap \\
minAddr, maxAddr : ADDR \\
voidMap, nvramMap, readOnlyMap, writeOnlyMap : \mathbb{P}\, ADDR \\
ram : \mathbb{P}\, ADDR \\
\hline
\langle voidMap, nvramMap, readOnlyMap, writeOnlyMap, ram \rangle \\
\quad \mathsf{partition}\ minAddr \mathinner{..} maxAddr \\
\hline
\end{array}
$$

The linker's type checking semantics are correct if the following conditions are satisfied.

$$
\begin{array}{|l}
\textit{LinkTypeOkay} \\
\hline
\textit{LinkDeclOkay} \\
hw : \textit{HwMap} \\
\hline
(\operatorname{ran} \circ \textit{second})(\mathcal{T}_L(M, cm)) = \varnothing \\
\mathsf{disjoint} \bigcup \circ \operatorname{ran} \circ \textit{atsMap} \\
\exists\, \textit{HwMap};\ \textit{map} : \mathbb{P}\, \textit{ATTR} \pfun \mathbb{P}\, \textit{ADDR} \mid \\
\qquad hw = \theta \textit{HwMap} \\
\qquad \land\ \textit{map} = \{\ A : \{\varnothing, \{\textit{nvram}\}, \{\textit{readOnly}\}, \{\textit{writeOnly}\}\} \bullet \\
\qquad\qquad A \mapsto \bigcup \{\ n : \operatorname{dom} \textit{atsMap} \bullet \textit{atsMap}\ n\ A\ \}\ \} \bullet \\
\quad \mathsf{disjoint} \langle \textit{voidMap}, \bigcup (\operatorname{ran} \textit{map}) \rangle \\
\quad \land\ \textit{map}\{\textit{nvram}\} \subseteq \textit{ram} \cup \textit{nvramMap} \\
\quad \land\ \textit{map}\{\textit{readOnly}\} \subseteq \textit{ram} \cup \textit{readOnlyMap} \\
\quad \land\ \textit{map}\{\textit{writeOnly}\} \subseteq \textit{ram} \cup \textit{writeOnlyMap} \\
\quad \land\ \textit{map}\ \varnothing \subseteq \textit{ram} \cup \textit{nvramMap} \\
\end{array}
$$

The conditions ensure that:

- the linked modules type-check correctly
- no 'AT' locations overlap between modules (that the locations do not overlap within a module is checked at compile time)
- that memory attributes are assigned to appropriate physical memory types:
  - no variables are assigned to *void* loactions
  - *nvram* variables are assigned to *ram* or *nvram* memory
  - *readOnly* variables are assigned to *ram* or *readOnly* memory
  - *writeOnly* variables are assigned to *ram* or *writeOnly* memory
  - 'AT' variables with no requested memory type are assigned to *ram* or *nvram* memory

6 Linker dynamic semantics

6.1 Phases

The dynamic semantics can be defined in a number of phases.

6.1.0.1 Make all labels unique. Each module has unique label numbering, but the labels numbers have to be made unique across all the modules

6.1.0.2 Flatten. All the labels are now unique, and so the sequence of module segments can be flattened, and concatenated to the main code segment.

6.1.0.3 Replace X instructions. All the *xProc* and *xCall* XAspAL instructions have to be replaced with AspAL instructions.

6.1.0.4 Jump to start of code. If there is any unsigned library code present, a jump around this to the first module is needed.

All of these phases can be incorporated into a single pass by the linker.

6.2 Label environment

EnvL is a label environment, used to map procedures and functions to their corresponding unique labels.

$$EnvL == Env[LABEL]$$

6.3 Update *X_DATA_INSTR*

upddata updates a particular *X_DATA_INSTR* with the new memory offset required, so that all the instructions use fixed addressing. This takes an

XAspAL instruction to a *AspAL* instruction that can be understood by the hexer.

$$
\begin{array}{|l}
updata : X\_DATA\_INSTR \times ADDR \times ADDR \times \mathbb{N} \pfun X\_INSTR \\
\hline
\forall\, dinstr : X\_DATA\_INSTR;\ start, \delta : ADDR;\ size : \mathbb{N} \bullet \\
\quad updata(xlrda, \delta, start, size) = xData(xlada, \delta + start) \\
\quad \land\ updata(xsdro, \delta, start, size) = xData(xsdr, topop + \delta) \\
\quad \land\ updata(xsdrs, \delta, start, size) = xData(xsdr, start + size - 2 - \delta) \\
\quad \land\ updata(xsdrh, \delta, start, size) = xData(xsdr, start + \delta) \\
\quad \land\ updata(xlada, \delta, start, size) = xData(xlada, \delta) \\
\quad \land\ updata(xsdr, \delta, start, size) = xData(xsdr, \delta)
\end{array}
$$

updparam updates a single actual parameter call instruction.

$$
\begin{array}{|l}
updparam : ID \times ID \times ID \pfun X\_INSTR \\
\hline
\forall\, \xi, \xi', \xi f : ID \bullet \\
\quad \exists\, \xi m : ID;\ \rho ot : EnvOTrace;\ \mu o : MO;\ l : LOCN \mid \\
\qquad (\xi f \notin \mathrm{dom}(\rho \delta t0\ \xi) \Rightarrow \xi m = \xi) \\
\qquad \land\ (\xi f \in \mathrm{dom}(\rho \delta t0\ \xi) \Rightarrow \rho \delta t0\ \xi m\ \xi f = checkOK) \\
\qquad \land\ \rho ot = compEnv\ \xi m \land \mu o = compMap\ \xi m \\
\qquad \land\ l = opLoc^{\sim}(\rho ot\ \xi f\ \xi')\langle\ \rangle \bullet \\
\quad updparam(\xi, \xi f, \xi') = \\
\qquad xData(xsdr, modStart\ \xi m + head(\mu o\ l))
\end{array}
$$

6.4 Update *X_INSTR*

updsngl takes a single XAspAL *X_INSTR* and converts it to the corresponding *AspAL* instruction (still an *X_INSTR*). It ensures that all the labels are unique by adding the accumulated label offset value n (in *SizeAddr*) of all the modules processed so far to any label instructions in this module. It also maintains the highest numbered label defined in the module, m in *LabelEnv*, as well as the mapping of the module's procedure and function names to their unique labels. The ρl label environment is the combination of exports already defined, overridden by any local procedure and function definitions. This mapping provides all the information required to replace any *xCall* call in the module.

$$
\begin{array}{|l}
\textit{LabelEnv} \\
\hline
m : \mathbb{N} \\
\rho l : \textit{EnvL} \\
\textit{used} : \mathbb{P}\, \textit{LABEL} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{SizeAddrs} \\
\hline
\xi : \textit{ID} \\
n, \textit{size} : \mathbb{N} \\
\textit{start} : \textit{ADDR} \\
\hline
\end{array}
$$

6.4.1 Update *xData*

An *xData* instruction is updated by replacing any relative addressing with absolute addressing.

$$
\begin{array}{|l}
\textit{updsngl} : \textit{X\_INSTR} \times \textit{LabelEnv} \times \textit{SizeAddrs} \pfun \\
\qquad\qquad \operatorname{seq} \textit{X\_INSTR} \times \textit{LabelEnv} \\
\hline
\forall\, i : \textit{X\_INSTR};\ \textit{LabelEnv};\ \textit{SizeAddrs} \mid i \in \operatorname{ran} \textit{xData} \bullet \\
\quad \exists\, di : \textit{X\_DATA\_INSTR};\ \delta : \textit{ADDR};\ I' : \operatorname{seq} \textit{X\_INSTR} \mid \\
\qquad\qquad i = \textit{xData}(di, \delta) \\
\qquad\qquad \land\ I' = \langle \textit{upddata}(di, \delta, \textit{start}, \textit{size}) \rangle \bullet \\
\qquad \textit{updsngl}(i, \theta \textit{LabelEnv}, \theta \textit{SizeAddrs}) = (I', \theta \textit{LabelEnv}) \\
\end{array}
$$

6.4.2 Update *xProc*

An *xProc* instruction is replaced by an *xLabel* instruction, identifying the instruction as the start of the routine.

$$
\begin{array}{l}
\forall\, i : X\_INSTR;\ LabelEnv;\ SizeAddrs \mid i \in \operatorname{ran} xProc \bullet \\
\quad \exists\, \xi' : ID;\ l : LABEL;\ I' : \operatorname{seq} X\_INSTR;\ LabelEnv' \mid \\
\qquad i = xProc(\xi', l) \\
\qquad \land\ m' = max\ \{m, l\} \\
\qquad \land\ \rho l' = \rho l \oplus \{\xi' \mapsto (l + n)\} \\
\qquad \land\ used' = used \\
\qquad \land\ I' = \langle xLabel\ (xli, l + n)\rangle \bullet \\
\quad updsngl(i, \theta LabelEnv, \theta SizeAddrs) = (I', \theta LabelEnv')
\end{array}
$$

6.4.3 Update *xLabel*

An *xLabel* instruction where the label is unique relative to the module is updated with a label that is globally unique.

$$
\begin{array}{l}
\forall\, i : X\_INSTR;\ LabelEnv;\ SizeAddrs \mid i \in \operatorname{ran} xLabel \bullet \\
\quad \exists\, li : X\_LABEL\_INSTR;\ l : LABEL;\ I' : \operatorname{seq} X\_INSTR;\ LabelEnv' \mid \\
\qquad i = xLabel(li, l) \\
\qquad \land\ m' = \textbf{if}\ li = xli\ \textbf{then}\ max\{m, l\}\ \textbf{else}\ m \\
\qquad \land\ \rho l' = \rho l \\
\qquad \land\ used' = used \\
\qquad \land\ I' = \langle xLabel(li, l + n)\rangle \bullet \\
\quad updsngl(i, \theta LabelEnv, \theta SizeAddrs) = (I', \theta LabelEnv')
\end{array}
$$

6.4.4 Update *xBvar*

An *xBvar* instruction is replaced by an *xData* instruction referencing the appropriate variable in the exporting module.

$$
\begin{array}{l}
\forall\, i : X\_INSTR;\ LabelEnv;\ SizeAddrs \mid i \in \operatorname{ran} xBvar \bullet \\
\quad \exists\, \xi', \xi m : ID;\ N : \operatorname{seq} \mathbb{N};\ A : \mathbb{P}\, ATTR; \\
\qquad\qquad \rho ot : EnvOTrace;\ \mu o : MO;\ l : LOCN;\ \delta : ADDR; \\
\qquad\qquad I' : \operatorname{seq} X\_INSTR \mid \\
\qquad\quad i = xBvar(\xi', N) \land \xi m \in \operatorname{dom} \rho\delta t0 \\
\qquad\quad \land\ \xi' \in \operatorname{dom}(\rho\delta t0\ \xi m) \\
\qquad\quad \land\ A = (variable^{\sim}\ (\rho\tau t0\ \xi m\ \xi')).A \\
\qquad\quad \land\ \rho ot = compEnv\ \xi m \land \mu o = compMap\ \xi m \\
\qquad\quad \land\ l = opLoc^{\sim}(\rho ot\ \xi m\ \xi')N \land \delta = head(\mu o\ l) \\
\qquad\quad \land\ I' = \langle xData(xlada, \\
\qquad\qquad\qquad \textbf{if}\ A \cap \operatorname{ran} dataAt = \varnothing \\
\qquad\qquad\qquad\quad \textbf{then}\ modStart\ \xi m + \delta\ \textbf{else}\ \delta)\rangle \bullet \\
\qquad updsngl(i, \theta LabelEnv, \theta SizeAddrs) = (I', \theta LabelEnv)
\end{array}
$$

6.4.5 Update *xConst*

An *xConst* instruction is replaced by load instructions that load the value of the appropriate constant in the exporting module.

$$
\begin{array}{l}
\forall\, i : X\_INSTR;\ LabelEnv;\ SizeAddrs \mid i \in \operatorname{ran} xConst \bullet \\
\quad \exists\, \xi', \xi m : ID;\ \rho ot : EnvOTrace;\ \kappa : VALUE;\ I' : \operatorname{seq} X\_INSTR \mid \\
\qquad\quad i = xConst\xi' \land \xi m \in \operatorname{dom} \rho\delta t0 \\
\qquad\quad \land\ \xi' \in \operatorname{dom}(\rho\delta t0\ \xi m) \\
\qquad\quad \land\ \rho ot = compEnv\ \xi m \\
\qquad\quad \land\ \kappa = opExport^{\sim}(\rho ot\ \xi m\ \xi') \\
\qquad\quad \land\ I' = \mathcal{O}_{l\tau i}(constType\ \kappa, number\ \kappa) \bullet \\
\qquad updsngl(i, \theta LabelEnv, \theta SizeAddrs) = (I', \theta LabelEnv)
\end{array}
$$

6.4.6 Update *xCall*

The *xCall* instruction is replaced by an *xrg* goto instruction, with a label identifying the routine to go to. The routine could be internal or external to the module. If a procedure or function is declared locally within a module, then it cannot also be imported to that module, and so the call must be

internal. Otherwise it is external.

$$\begin{array}{|l}
\forall\, i : X_INSTR;\ LabelEnv;\ SizeAddrs \mid i \in \operatorname{ran} xCall \bullet \\
\quad \exists\, \xi' : ID;\ I' : \operatorname{seq} X_INSTR \mid \\
\qquad\qquad i = xCall\xi' \\
\qquad\qquad \land\ I' = \langle xLabel(xrg, \rho l\ \xi') \rangle \bullet \\
\qquad\quad updsngl(i, \theta LabelEnv, \theta SizeAddrs) = (I', \theta LabelEnv) \\
\end{array}$$

6.4.7 Update *xCallOp*

An *xCallOp* instruction (optimised operator call) is replaces by an *xrg* goto instruction to that label.

$$\begin{array}{|l}
\forall\, i : X_INSTR;\ LabelEnv;\ SizeAddrs \mid i \in \operatorname{ran} xCallOp \bullet \\
\quad \exists\, l : LABEL;\ I' : \operatorname{seq} X_INSTR;\ LabelEnv' \mid \\
\qquad\qquad i = xCallOpl \land m' = m \land \rho l' = \rho l \\
\qquad\qquad \land\ used' = used \cup \{l\} \\
\qquad\qquad \land\ I' = \langle xLabel(xrg, l) \rangle \bullet \\
\qquad\quad updsngl(i, \theta LabelEnv, \theta SizeAddrs) = (I', \theta LabelEnv') \\
\end{array}$$

6.4.8 Update *xPa*

An *xPa* instruction is replaced by the appropriate actual parameter call instruction.

$$\begin{array}{|l}
\forall\, i : X_INSTR;\ LabelEnv;\ SizeAddrs \mid i \in \operatorname{ran} xPa \bullet \\
\quad \exists\, \xi', fn : ID;\ I' : \operatorname{seq} X_INSTR \mid \\
\qquad\qquad i = xPa(fn, \xi') \\
\qquad\qquad \land\ I' = \langle updparam(\xi, fn, \xi') \rangle \bullet \\
\qquad\quad updsngl(i, \theta LabelEnv, \theta SizeAddrs) = (I', \theta LabelEnv) \\
\end{array}$$

6.4.9 Update *xIndirect*

The remaining instructions are unchanged.

$$\forall\, i : X\_INSTR;\ LabelEnv;\ SizeAddrs \mid i \in \operatorname{ran} xIndirect \cup \operatorname{ran} indirect \cup \operatorname{ran} immediate \bullet updsngl(i, \theta LabelEnv, \theta SizeAddrs) = (\langle i \rangle, \theta LabelEnv)$$

6.5 Update seq X_INSTR

The *updseq* function takes a sequence of instructions and applies the *updsngl* function to each instruction in order.

$$updseq : \operatorname{seq} X\_INSTR \times LabelEnv \times SizeAddrs \pfun \operatorname{seq} X\_INSTR \times LabelEnv$$

$$\forall\, i : X\_INSTR;\ LabelEnv;\ SizeAddrs \bullet updseq(\langle i \rangle, \theta LabelEnv, \theta SizeAddrs) = updsngl(i, \theta InstrLabel, \theta SizeAddrs)$$

$$\forall\, i : X\_INSTR;\ I : \operatorname{seq} X\_INSTR;\ LabelEnv;\ SizeAddrs \bullet \exists\, I', I'' : \operatorname{seq} X\_INSTR;\ LabelEnv';\ LabelEnv'' \mid (I', \theta LabelEnv') = updsngl(i, \theta LabelEnv, \theta SizeAddrs) \land (I'', \theta LabelEnv'') = updseq(I, \theta LabelEnv', \theta SizeAddrs) \bullet updseq(\langle i \rangle \frown I, \theta LabelEnv, \theta SizeAddrs) = (I' \frown I'', \theta LabelEnv'')$$

6.6 Process module

The final processing required is to handle the application of *updseq* one module at a time from the sequence of modules.

Before the module is processed, the input parameter *LabelEnv* contains:

- m, the highest label number used so far, in previous modules
- ρl, the export identifiers defined so far, mapped to their unique labels
- *used*, the labels used so far, in previous modules

At the end of the module processing, the output contains :

- m' is the highest (unlinked) label found in the module just processed, so $m' + m$ is the new highest unique value.
- $\rho l''$, the mapping of all exports so far, with any local definitions. ρl is updated with any exports from this module. This set of exports is found from the *EnvDL* linker environment, which maintains all the imports and exports for each module. $\rho l''$ is restricted to this set, and combined with the original ρl.
- $used'$, the labels used so far, in this and previous modules

$$
\begin{array}{|l}
processM : CompModule \times ADDR \times LabelEnv \pfun \\
\qquad \seq X\_INSTR \times LabelEnv \\
\hline
\forall\, CompModule;\ start : ADDR;\ LabelEnv \bullet \\
\quad \exists\, InstrLabel';\ \rho l'' : EnvL \mid \\
\qquad (I', \theta LabelEnv') = \\
\qquad\quad updseq(I, \lbag\, m == 0, \rho l == \rho l, used == used \,\rbag, \\
\qquad\qquad \lbag\, \xi == \xi, n == m, size == size, start == start \,\rbag) \\
\qquad \land\ \rho l'' = \rho l \cup (\dom(\rho\delta t0\ \xi \rhd \{checkOK\}) \lhd \rho l') \bullet \\
\quad processM(\theta CompModule, start, \theta LabelEnv) = \\
\qquad (I', \lbag\, m == m' + m, \rho l == \rho l'', used == used' \,\rbag)
\end{array}
$$

The extension to a sequence of modules is the standard inductive definition.

$$
\begin{array}{|l}
processMs : \seq CompModule \times \seq ADDR \times LabelEnv \pfun \\
\qquad \seq X\_INSTR \times LabelEnv \\
\hline
\forall\, LabelEnv \bullet processMs(\langle\,\rangle, \langle\,\rangle, \theta LabelEnv) = (\langle\,\rangle, \theta LabelEnv) \\
\forall\, cm : CompModule;\ M : \seq CompModule;\ \delta : ADDR; \\
\qquad \Delta : \seq ADDR;\ LabelEnv \bullet \\
\quad \exists\, I', I'' : \seq X\_INSTR;\ LabelEnv';\ LabelEnv'' \mid \\
\qquad (I', \theta LabelEnv') = processM(cm, \delta, \theta LabelEnv) \\
\qquad \land\ (I'', \theta LabelEnv'') = processMs(M, \Delta, \theta LabelEnv') \bullet \\
\quad processMs(\langle cm\rangle \frown M, \langle\delta\rangle \frown \Delta, \theta LabelEnv) = \\
\qquad (I' \frown I'', \theta LabelEnv'')
\end{array}
$$

7 Actual Linker

7.1 Optimised operators

genasp conditionally produces code for unsigned multiplication and division, only if they are used in the program.

$$
\begin{array}{|l}
genasp : \mathbb{P}\, LABEL \pfun \operatorname{seq} X\_INSTR \\
\hline
\forall\, used : \mathbb{P}\, LABEL \bullet \\
\quad \exists\, I, I' : \operatorname{seq} X\_INSTR \mid \\
\qquad I = \textbf{if } binopfn\ umul \in used \textbf{ then } \mathcal{O}_{UMUL} \textbf{ else } \langle\,\rangle \\
\qquad \land\ I' = \textbf{if } binopfn\ udiv \in used \textbf{ then } \mathcal{O}_{UDIV} \textbf{ else } \langle\,\rangle \bullet \\
\quad genasp\ used = I \frown I'
\end{array}
$$

7.2 Linking

The linker takes a sequence of modules and a main module, a hardware memory map, and the size of the operator stack, and returns the linked sequence of instructions.

$$
\begin{array}{|l}
link : (\operatorname{seq} CompModule \times CompModule) \times HwMap \times \mathbb{N} \pfun \\
\qquad \operatorname{seq} X\_INSTR \times MO \\
\hline
\forall\, M : \operatorname{seq} CompModule;\ cm : CompModule;\ hw : HwMap;\ sizetopop : \mathbb{N} \mid \\
\qquad LinkTypeOkay \bullet \\
\quad \exists\, M' : \operatorname{seq}_1 CompModule;\ ats : \mathbb{P}\, ADDR;\ F, F', F'' : Free; \\
\qquad \Delta : \operatorname{seq}_1 ADDR;\ I : \operatorname{seq} X\_INSTR;\ LabelEnv; \\
\qquad sizetopop' : \mathbb{N};\ I'', Iop : \operatorname{seq} X\_INSTR; \\
\qquad \xi : ID;\ \mu o : MO \mid \\
\qquad M' = M \frown \langle cm \rangle \\
\qquad \land\ ats = \bigcup(\operatorname{ran}(\bigcup(\operatorname{ran} atsMap))) \\
\qquad \land\ F = blockify(hw.ram \setminus ats) \\
\qquad \land\ (F', topop) = assignTopOp(F, sizetopop) \\
\qquad \land\ (F'', \Delta) = assignMod(F', getModSize\ M') \\
\qquad \land\ 0 \notin \{topop\} \cup \operatorname{ran} \Delta \\
\qquad \land\ modStart = \{\ n : \operatorname{dom} M' \bullet (M'\ n).\xi \mapsto \Delta\ n\ \} \\
\qquad \land\ compEnv = \{\ n : \operatorname{dom} M' \bullet (M'\ n).\xi \mapsto (M'\ n).\rho ot\ \} \\
\qquad \land\ compMap = \{\ n : \operatorname{dom} M' \bullet (M'\ n).\xi \mapsto (M'\ n).\mu o\ \} \\
\qquad \land\ (I, \theta LabelEnv) = \\
\qquad\quad processMs(M', \Delta, \\
\qquad\qquad (\!|\ m == firstAvailableLabel, \rho l == \varnothing, used == \varnothing\ |\!)) \\
\qquad \land\ Iop = genasp\ used \\
\qquad \land\ (Iop = \langle\,\rangle \Rightarrow I'' = \langle\,\rangle) \\
\qquad \land\ (Iop \neq \langle\,\rangle \Rightarrow \\
\qquad\quad (\ \exists\, CompModule;\ I' : \operatorname{seq} X\_INSTR;\ LabelEnv' \mid \\
\qquad\qquad I = Iop \bullet \\
\qquad\qquad (I', \theta LabelEnv') = \\
\qquad\qquad\quad processM(\theta CompModule, 0, \\
\qquad\qquad\qquad (\!|\ m == 0, \rho l == \varnothing, used == used\ |\!)) \\
\qquad\qquad \land\ (\ \exists\, l : LABEL \setminus used' \bullet \\
\qquad\qquad\quad I'' = \langle xLabel(xrg, l)\rangle \frown I' \frown \langle xLabel(xli, l)\rangle\ )\ )) \\
\qquad \land\ \operatorname{dom} \mu o = \operatorname{dom}(\bigcup(\operatorname{ran} compMap)) \\
\qquad \land\ (\ \forall \xi : \operatorname{dom} compMap;\ l : LOCN \mid l \in \operatorname{dom}(compMap\ \xi) \bullet \\
\qquad\quad \mu o\ l = \textbf{if } \operatorname{ran}(compMap\ \xi\ l) \subseteq ats \\
\qquad\qquad \textbf{then } compMap\ \xi\ l \\
\qquad\qquad \textbf{else } \{\ n : \operatorname{dom}(compMap\ \xi\ l) \bullet \\
\qquad\qquad\qquad n \mapsto (modStart\ \xi + compMap\ \xi\ l\ n)\ \}\ ) \bullet \\
\quad link((M, cm), hw, sizetopop) = (I'' \frown I, \mu o)
\end{array}
$$

The actual linker can be considered in three parts.

1. *LinkTypeOkay* : The symbol declaration semantics, type checking semantics, and 'AT' allocations, are checked to be satisfactory.
2. Memory locations for 'AT' variables, the operator stack, and each module's heap and stack, are assigned within the memory map. (The remaining sequence of free blocks is given by F''.)
3. If a successful memory model can be determined (no addresses equal to zero), the *processMs* function is applied, to perform the linking.
4. The last two predicates give the value of the global linked memory map; they are provided for the proof, and do not need to be implemented.

The output is the sequence of AspAL instructions produced.

The instruction processing is done in a single pass through the compiled sequence of *X_INSTR*. The linker starts by calling *processMs*, with the sequence of start addresses for each module Δ, the value of *topop*, an initial label of *firstAvailableLabel* (the value of *firstAvailableLabel* is set to allow space for the optimised operators' labels, so that they are unique), an initial empty sequence of instructions, and an initial empty label map. If there are any instructions for the unsigned operators, $Iop \neq \langle\ \rangle$, a jump around them is added to the code.

Parent Section: AspAL Specification

8 Hexer

As detailed in the introduction section, the hexer transforms sequences of AspAL instructions into sequences of Asp instructions. This involves the removing of labelled instructions and replacing them with instructions using program instructions. The hexer operates by processing each AspAL instruction individually. Each generated Asp instruction is paired with a unique program address. This approach is used within the specification of the hexer, and at each stage the current program address and the sequence of processed Asp instructions is maintained.

$$Map == LABEL \pfun PROG\_ADDRESS$$
$$SubProg == \mathrm{seq}(PROG\_ADDRESS \times INSTR)$$

8.1 Semantics

The function map_0 defines the mapping from labels to absolute program addresses.

$$map_0 : Map$$

The function is built up as the sequence of AspAL instructions is processed. Whenever a label instruction is encountered the appropriate program address is stored for the label.

The initial program address when starting a program is *StartProg*. Currently this is set to 0.

$$StartProg == 0$$

8.2 Hexing AspAL instructions

Converting *xIndirect* instruction: the indirect absolute goto label instruction converts into the corresponding Asp instruction `pab`.

$$
\begin{array}{|l}
hex : X\_INSTR \pfun \\
\quad Map \times PROG\_ADDRESS \times SubProg \pfun \\
\quad\quad Map \times PROG\_ADDRESS \times SubProg \\
\hline
\forall\, m : Map;\ \pi : PROG\_ADDRESS;\ S : SubProg \bullet \\
\quad hex(xIndirect\ xag)(m, \pi, S) = \\
\quad\quad (m, \pi + 1, S \frown \langle(\pi, asp\_indirect\ \texttt{pab})\rangle)
\end{array}
$$

Converting *xData* instructions: these set the data register and the a and b registers to particular values. Conversion involves setting the appropriate registers with the low and high bytes of the supplied address.

$$
\begin{array}{|l}
hex : X\_INSTR \pfun \\
\quad Map \times PROG\_ADDRESS \times SubProg \pfun \\
\quad\quad Map \times PROG\_ADDRESS \times SubProg \\
\hline
\forall\, \delta : DATA\_ADDRESS;\ m : Map;\ \pi : PROG\_ADDRESS;\ S : SubProg \bullet \\
\quad hex(xData(xsdr, \delta))(m, \pi, S) = \\
\quad\quad (m, \pi + 2, S \\
\quad\quad\quad \frown \langle(\pi, both\_immediate(\texttt{dix}, hiByte\ \delta))\rangle \\
\quad\quad\quad \frown \langle(\pi + 1, both\_immediate(\texttt{dxi}, loByte\ \delta))\rangle) \\
\quad \land\ hex(xData(xlada, \delta))(m, \pi, S) = \\
\quad\quad (m, \pi + 2, S \\
\quad\quad\quad \frown \langle(\pi, both\_immediate(\texttt{aldi}, hiByte\ \delta))\rangle \\
\quad\quad\quad \frown \langle(\pi + 1, both\_immediate(\texttt{bldi}, loByte\ \delta))\rangle)
\end{array}
$$

Converting *xLabel* instructions: In each case (except the label declaration instruction *xli*) this involves using the program address corresponding to the label, found from the global map_0. This address is used to perform the required operation using Asp instructions. *xli* does not convert to Asp code itself; it updates the map.

$$
\begin{array}{l}
hex : X\_INSTR \pfun \\
\quad Map \times PROG\_ADDRESS \times SubProg \pfun \\
\quad\quad Map \times PROG\_ADDRESS \times SubProg \\
\hline
\forall\, l : LABEL;\ m : Map;\ \pi : PROG\_ADDRESS;\ S : SubProg \bullet \\
\quad \exists\, \delta : DATA\_ADDRESS \mid \delta = map_0\ l \bullet \\
\quad\quad hex(xLabel(xrja, l))(m, \pi, S) = \\
\quad\quad\quad (m, \pi + 1, S \\
\quad\quad\quad\quad \frown \langle(\pi, asp\_immediate(\texttt{cappi}, \delta - \pi))\rangle) \\
\quad\quad \land\ hex(xLabel(xrjb, l))(m, \pi, S) = \\
\quad\quad\quad (m, \pi + 1, S \\
\quad\quad\quad\quad \frown \langle(\pi, asp\_immediate\ (\texttt{cbppi}, \delta - \pi))\rangle) \\
\quad\quad \land\ hex(xLabel(xrjal, l))(m, \pi, S) = \\
\quad\quad\quad (m, \pi + 5, S \frown \langle(\pi, both\_indirect\ \texttt{nca})\rangle \\
\quad\quad\quad\quad \frown \langle(\pi + 1, asp\_immediate(\texttt{cappi}, 4))\rangle \\
\quad\quad\quad\quad \frown \langle(\pi + 2, both\_immediate(\texttt{aldi}, hiByte\ \delta))\rangle \\
\quad\quad\quad\quad \frown \langle(\pi + 3, both\_immediate(\texttt{bldi}, loByte\ \delta))\rangle \\
\quad\quad\quad\quad \frown \langle(\pi + 4, asp\_indirect\ \texttt{pab})\rangle) \\
\quad\quad \land\ hex(xLabel(xrjbl, l))(m, \pi, S) = \\
\quad\quad\quad (m, \pi + 5, S \frown \langle(\pi, both\_indirect\ \texttt{ncb})\rangle \\
\quad\quad\quad\quad \frown \langle(\pi + 1, asp\_immediate(\texttt{cbppi}, 4))\rangle \\
\quad\quad\quad\quad \frown \langle(\pi + 2, both\_immediate(\texttt{aldi}, hiByte\ \delta))\rangle \\
\quad\quad\quad\quad \frown \langle(\pi + 3, both\_immediate(\texttt{bldi}, loByte\ \delta))\rangle \\
\quad\quad\quad\quad \frown \langle(\pi + 4, asp\_indirect\ \texttt{pab})\rangle) \\
\quad\quad \land\ hex(xLabel(xrg, l))(m, \pi, S) = \\
\quad\quad\quad (m, \pi + 3, S \\
\quad\quad\quad\quad \frown \langle(\pi, both\_immediate(\texttt{aldi}, hiByte\ \delta))\rangle \\
\quad\quad\quad\quad \frown \langle(\pi + 1, both\_immediate(\texttt{bldi}, loByte\ \delta))\rangle \\
\quad\quad\quad\quad \frown \langle(\pi + 2, asp\_indirect\ \texttt{pab})\rangle) \\
\quad\quad \land\ hex(xLabel(xrgs, l))(m, \pi, S) = \\
\quad\quad\quad (m, \pi + 1, S \\
\quad\quad\quad\quad \frown \langle(\pi, asp\_immediate(\texttt{ppi}, \delta - \pi))\rangle) \\
\quad\quad \land\ hex(xLabel(xll, l))(m, \pi, S) = \\
\quad\quad\quad (m, \pi + 2, S \\
\quad\quad\quad\quad \frown \langle(\pi, both\_immediate(\texttt{aldi}, hiByte\ \delta))\rangle \\
\quad\quad\quad\quad \frown \langle(\pi + 1, both\_immediate(\texttt{bldi}, loByte\ \delta))\rangle) \\
\forall\, l : LABEL;\ m : Map;\ \pi : PROG\_ADDRESS;\ S : SubProg \bullet \\
\quad hex(xLabel(xli, l))(m, \pi, S) = (m \cup \{l \mapsto \pi\}, \pi, S)
\end{array}
$$

8.3 Hexing common instructions

This deals with the common Asp and AspAL instructions.

$$
\begin{array}{l}
hex : X\_INSTR \pfun \\
\qquad Map \times PROG\_ADDRESS \times SubProg \pfun \\
\qquad\qquad Map \times PROG\_ADDRESS \times SubProg \\
\hline
\forall\, i : IMMED\_INSTR \times BYTE;\ m : Map;\ \pi : PROG\_ADDRESS; \\
\qquad\qquad S : SubProg \bullet \\
\qquad hex(immediate\ i)(m, \pi, S) = \\
\qquad\qquad (m, \pi + 1, S \frown \langle (\pi, both\_immediate\ i) \rangle) \\
\forall\, i : INDIRECT\_INSTR;\ m : Map;\ \pi : PROG\_ADDRESS; \\
\qquad\qquad S : SubProg \bullet \\
\qquad hex(indirect\ i)(m, \pi, S) = \\
\qquad\qquad (m, \pi + 1, S \frown \langle (\pi, both\_indirect\ i) \rangle)
\end{array}
$$

8.4 Multiple instruction processing

The *hexS* function handles sequences of AspAL instructions. This calls the *hex* function for each instruction and supplies the output to the next call.

$$
\begin{array}{l}
hexS : \mathrm{seq}\, X\_INSTR \pfun \\
\qquad Map \times PROG\_ADDRESS \times SubProg \pfun \\
\qquad\qquad Map \times PROG\_ADDRESS \times SubProg \\
\hline
\forall\, m : Map;\ \pi : PROG\_ADDRESS;\ S : SubProg \bullet \\
\qquad hexS\langle\ \rangle(m, \pi, S) = (m, \pi, S) \\
\forall\, I : \mathrm{seq}\, X\_INSTR;\ i : X\_INSTR;\ m : Map; \\
\qquad\qquad \pi : PROG\_ADDRESS;\ S : SubProg \bullet \\
\qquad hexS(\langle i \rangle \frown I)(m, \pi, S) = hexS\ I\ (hex\ i\ (m, \pi, S))
\end{array}
$$

8.5 Hexer implementation

The hexer operates by calling the *hexS* function with an initial empty map, the starting program address, and the initial empty Asp sequence. The

generated pairs of program addresses and Asp instructions is the output. (Note that this definition also sets the value of map_0.)

$$
\begin{array}{|l}
hexer : \mathrm{seq}_1\, X\_INSTR \rightarrow SubProg \\
\hline
\forall\, I : \mathrm{seq}_1\, X\_INSTR \bullet \\
\quad \exists\, \pi : PROG\_ADDRESS;\ S : SubProg \mid \\
\qquad (map_0, \pi, S) = hexS\ I(\varnothing, StartProg, \langle\ \rangle) \bullet \\
\quad hexer\ I = S \\
\end{array}
$$

References

[Formaliser] Susan Stepney. *The Formaliser Manual*. Logica.

[ISO-Z] ISO/IEC 13568. *Information Technology—Z Formal Specification Notation—Syntax, Type System and Semantics*. 2002.

[Spivey 1992] J. Michael Spivey. *The Z Notation: a Reference Manual*. Prentice Hall, 2nd edition, 1992.

[Stepney *et al.* 1991] Susan Stepney, Dave Whitley, David Cooper, and Colin Grant. A demonstrably correct compiler. *BCS Formal Aspects of Computing*, 3:58–101, 1991.

[Stepney 1993] Susan Stepney. *High Integrity Compilation: A Case Study*. Prentice Hall, 1993.

[Stepney 1998] Susan Stepney. Incremental development of a high integrity compiler: experience from an industrial development. In *Third IEEE High-Assurance Systems Engineering Symposium (HASE'98), Washington DC*, 1998.

[Stringer-Calvert *et al.* 1997] David W.J. Stringer-Calvert, Susan Stepney, and Ian Wand. Using PVS to prove a Z refinement: A case study. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME '97: Formal Methods: Their Industrial Application and Strengthened Foundations*, number 1313 in Lecture Notes in Computer Science, pages 573–588. Springer Verlag, September 1997.

Index